Password-Authenticated Group Key Agreement with Adaptive Security and Contributiveness

Michel Abdalla, Dario Catalano, Céline Chevalier, David Pointcheval

This is the full version of the extended abstract which appears in Africacrypt '09 (21 – 25 june 2009, Gammarth, Tunisia)
B. Preneel Ed., Springer-Verlag, LNCS ????, pages ???–???.

Password-Authenticated Group Key Agreement with Adaptive Security and Contributiveness

Michel Abdalla [1], Dario Catalano [2], Céline Chevalier [1], and David Pointcheval [1]

[1] École Normale Supérieure, CNRS-INRIA, Paris, France
[2] Università di Catania, Catania, Italy

Abstract. Adaptively-secure key exchange allows the establishment of secure channels even in the presence of an adversary that can corrupt parties adaptively and obtain their internal states. In this paper, we give a formal definition of contributory protocols and define an ideal functionality for password-based group key exchange with explicit authentication and contributiveness in the UC framework. As with previous definitions in the same framework, our definitions do not assume any particular distribution on passwords or independence between passwords of different parties. We also provide the first steps toward realizing this functionality in the above strong adaptive setting by analyzing an efficient existing protocol and showing that it realizes the ideal functionality in the random-oracle and ideal-cipher models based on the CDH assumption.

1 Introduction

Motivation. The main goal of an authenticated key exchange (AKE) protocol is to allow users to establish a common key over a public channel, even in the presence of adversaries. The most common way to achieve this goal is to rely either on a public-key infrastructure (PKI) or on a common high-entropy secret key [38]. Unfortunately, these methods require the existence of trusted hardware capable of storing high-entropy secret keys. In this paper, we focus on a different and perhaps more realistic scenario where secret keys are assumed to be short passwords.

Since the seminal work by Bellovin and Merritt [11], password-based key exchange has become quite popular. Due to their low entropy, passwords are easily memorizable by humans and avoid the need for trusted hardware. On the other hand, the low entropy of passwords makes them vulnerable to exhaustive search: perfect forward secrecy thus becomes quite important. This notion means that, even if the password of a user is later guessed or leaked, keys established before the leakage of the password remain private. Depending on the security model, the privacy of the common key has been modeled via either "semantic security" [9] or the indistinguishability of the actual protocol and an ideal one [26]. In both cases, the leakage of the long-term secret (password in our case) has been modeled by "corruption" queries. Unfortunately, the long-term secret may not be the only information leaked during a corruption. Ephemeral secret leakage has also been shown to cause severe damages in some contexts [36, 35], and thus "strong corruptions" should also be considered [39]. However, it may not be very realistic to allow the designer of a protocol to decide which information is revealed by such a *strong corruption* query. The universal composability (UC) framework [24], on the other hand, allows for a different approach: whenever a strong corruption occurs, the adversary breaks into the corrupted players, learns whatever information is required to complete the session, and controls the player thereafter. This seems to be a more realistic scenario.

In this paper, we consider stronger corruptions for password-based group key exchange protocols in the adaptive setting, in which adversaries are allowed to corrupt parties adaptively based on the information they have gathered so far. Despite numerous works on group key exchange protocols, very few schemes have been proven to withstand strong corruptions in case of adaptive adversaries. In the context of group AKE, Katz and Shin [34] proposed a compiler that converts any protocol that is secure in a weak corruption model into one that is secure in a strong corruption model, but their protocol relies on signatures and does not work in the password-based scenario. In the case of password-based group AKE protocols, to the best of our knowledge, the only scheme shown to withstand adaptive corruptions

is due to Barak *et al.* [6] using general techniques from multi-party computation. Unfortunately, their scheme is not practical.

Another issue being considered here is contributiveness [13, 28, 21]. In the security model of Katz and Shin [34], one just wants to prevent the adversary from fully determining the key, unless it has corrupted one player in the group. However, it would be better to ensure the randomness of the key, unless it has compromised the security of a sufficient number of players: If the adversary has not compromised too many players, it should not be able to bias the key. There are several advantages in adopting a stronger notion of contributiveness for group key exchange protocols. First, it creates a clear distinction between key distribution and key agreement protocols by describing in concrete terms the notion that, in a group key agreement protocol, each player should contribute equally to the session key. Second, it renders the protocol more robust to failures in that the session key is guaranteed to be uniformly distributed even if some of the players do not choose their contributions correctly (due to hardware malfunction, for instance). Third, it avoids scenarios in which malicious insiders secretly leak the value of a session key to a third party by either imposing specific values for the session key or biasing the distribution of the latter, so that the key can be quickly derived, and then the communication eavesdropped in real-time. For instance, we can imagine the third party to be an intelligence agency, such as the CIA, and the malicious insiders to be malicious pieces of hardware and software (pseudo-random generator) installed by the intelligence agency. Finally, if the absence of subliminal channels [40] can be guaranteed during the lifetime of the session key starting from the moment at which the players have gathered enough information to compute this key (a property that needs to be studied independently), then no malicious insider would be able to help an outsider to eavesdrop on the communication in real-time. Interestingly, since all the functions used in the later rounds are deterministic, this property seems to be satisfied by our scheme. Of course, one cannot keep an insider from later revealing this key, but by then it might already be too late for this information to be useful. On the other hand, physical protections or network surveillance can ensure that no stream of data can be sent out by the players during a confidential discussion. Contributiveness and the absence of subliminal channels therefore guarantee no real-time eavesdropping, thus yielding a new security property. However, the study of subliminal channels is out of the scope of the present paper.

Contributions. There are three main contributions in this paper. First, we investigate a stronger notion of security for password-based group AKE. This is done by combining recent results on the topic of AKE in the UC framework, including the work of Canetti *et al.* [25], Katz and Shin [34], and Barak *et al.* [6]. The first one described the ideal functionality for the password-based AKE, and proved that a variant of the KOY/GL scheme [33, 31] securely realizes the functionality, only against static adversaries (no strong corruptions available during the protocol, but at the beginning only). Katz and Shin [34] provided the ideal functionality for the group AKE, and proved that the new, derived, security notion is actually stronger than the usual Bresson *et al.* one [20]. Furthermore they formalized a strong corruption model, where honest players may be compromised at any point during the execution of the protocol, and leak their long-term and ephemeral secrets (the entire internal state). Barak *et al.* [6] considered protocols for general multi-party computation in the absence of authenticated channels. In particular, they provided a general and conceptually simple solution (though inefficient) to a number of problems including password-based group AKE.

In Section 2, we propose an ideal functionality for password-based group AKE. Our new functionality guarantees mutual authentication, which explicitly ensures that, if a player accepts the session key, then all his intended partners have been involved in the protocol and obtained the key material. Note however that the adversary may modify subsequent flows and eventually make some of the players reject while others accept. The protocol also inevitably leaks some information to the adversary, who ends up learning whether the parties share a common secret key or not. Following the approach suggested in [25], we assume that the passwords are chosen by the environment, who then hands them to the parties as input. This is the strongest security model, since it does not assume any distribution on

passwords. Furthermore, it allows the environment to force players to run the protocol using different (possibly related) passwords. This is useful, for example, to model the situation where users mistype their passwords.

Vulnerability of the passwords (whose entropy may be low) is modeled via split functionalities [6] and `TestPwd` queries. The use of split functionalities captures the fact that the adversary can always partition the players into disjoint subgroups and engage in separate executions of the protocols with each of these subgroups, playing the role of the other players. This is because, in the absence of strong authentication mechanisms such as signatures, honest parties cannot distinguish the case in which they interact with each other from the case in which they interact with the adversary. The use of `TestPwd` queries captures the fact that, within a particular subgroup, the adversary may be able to further divide the set of honest users into smaller groups and locally test the passwords of these users. In fact, in most password-based protocols based on the Burmester-Desmedt group key exchange [23] such as the one by Abdalla *et al.* [2], the adversary can test the value of the password held by a user by simply playing the role of the neighbors of this user. We note, however, that even though `TestPwd` queries may be avoided depending on the particular protocol being considered, its use does not severely weaken the security model since the adversary is still limited to at most two password tests per honest user for each session in which it plays an active role, by splitting the group into subgroups of one honest user each. Hence, one just needs to add one more bit of entropy to the password as a countermeasure.

A second contribution of this paper is to strengthen the original security model of Katz and Shin [34] by incorporating the notion of contributiveness in the functionality described above. A protocol is said to be (t, n)-contributory if no adversary can bias the key as long as (strictly) less than t players in the group of n players have been corrupted. This is stronger than the initial Katz-Shin model, which prevents the adversary from choosing the key, but as long as there is no corrupted player in the game ($t = 1$), only. Of course, one cannot prevent an insider adversary from learning the key, and possibly revealing it or the communication itself to outside. But we may hope to prevent "subliminal" leakage of information that would allow eavesdropping in real-time.

Our last contribution is to show that a slight variant of the password-based group AKE protocol by Abdalla *et al.* [2] based on the Burmester-Desmedt protocol [22, 23] and briefly described in Section 3, securely realizes the new functionality, even against adaptive adversaries. The proof is given in Section 4 (the details can be found in Appendix C) and is in the ideal-tweakable-cipher and random-oracle models [8]. Even though, from a mathematical point of view, it would be preferable to have a proof in the standard model, we point out that the protocol presented here is the first group key exchange protocol realizing such strong security notions, namely adaptive security against strong corruptions in the UC framework and $(n/2, n)$-contributiveness. In addition, our protocol is quite efficient. We also provide a modification to achieve $(n - 1, n)$-contributiveness.

Related work. Since the seminal Diffie-Hellman key exchange protocol [29], several extensions of that protocol to the group setting were proposed in the literature [22, 5] without a formal security model. The first security model for group key exchange was proposed by Bresson *et al.* [20], who later extended it to dynamic and concurrent setting [15, 16], using the same framework as Bellare *et al.* [9, 10]. In the UC framework [24], the first security model in the group setting was proposed by Katz and Shin [34]. Their model is quite strong, allowing the adversary to corrupt players adaptively and learn their internal state. In the password-based scenario, most of the previous work focused on the 2-party case. The first security models to appear [7, 14] were based on the frameworks by Bellare *et al.* [9, 10] and by Shoup [39]. In the UC framework, the first ones to propose an ideal functionality for password-based AKE were Canetti *et al.* [25]. More recently, Abdalla *et al.* [3] showed that the 2-party password-based authenticated key exchange protocol in [18] is also secure in the UC model against adaptive adversaries assuming the random-oracle and ideal-cipher models. In the group password-based AKE scenario, there has been a few protocols proposed in the literature, from the initial work by Bresson *et al.* [17, 19] to the more recent proposals by Dutta and Barua [30], Abdalla *et al.* [1,

2, 4], and Bohli *et al.* [12]. However, none of them appear to satisfy the security requirements being considered in this paper.

2 Definition of Security

Notations. We denote by k the security parameter. An event is said to be negligible if it happens with probability less than the inverse of any polynomial in k. If G is a finite set, $x \stackrel{R}{\leftarrow} G$ indicates the process of selecting x uniformly and at random in G (thus we implicitly assume that G can be sampled efficiently).

The UC Framework. Throughout this paper we assume basic familiarity with the universal composability framework. The interested reader is referred to [24, 25] for details. The model considered in this paper is the UC framework with joint state proposed by Canetti and Rabin [27].

Adaptive Adversaries. In this paper, we consider adaptive adversaries which are allowed to arbitrarily corrupt players at any moment during the execution of the protocol, thus getting complete access to their internal memory. In a real execution of the protocol, this is modeled by letting the adversary $\mathcal{A}$ obtain the password and the internal state of the corrupted player. Moreover, $\mathcal{A}$ can arbitrarily modify the player's strategy. In an ideal execution of the protocol, the simulator $\mathcal{S}$ gets the corrupted player's password and has to simulate its internal state, in a way that remains consistent to what was already provided to the environment.

Contributory Protocols. In addition, we consider a stronger corruption model against insiders than the one proposed by Katz and Shin in [34], where one allows the adversary to choose the session key as soon as there is a corruption. On the contrary, we define here a notion of contributory protocol which guarantees the distribution of the session keys to be random as long as there are enough honest participants in the session: the adversary cannot bias the distribution unless it controls a large number of players. More precisely, we say that a protocol is (t, n)-contributory if the group consists of n people and if the adversary cannot bias the key as long as it has corrupted (strictly) less than t players. More concretely, we claim that our proposed protocol is $(n/2, n)$-contributory, which means that the adversary cannot bias the key as long as there are at least half honest players. We even show in Appendix B.2 that $(n-1, n)$-contributiveness can be fulfilled by running parallel executions of our protocol.

The Random Oracle and Ideal Tweakable Cipher. In [25], Canetti *et al.* showed that there doesn't exist any protocol that UC-emulates the two-party password-based key-exchange functionality in the plain model (i.e. without additional setup assumptions). Here we show how to securely realize a similar functionality without setup assumption but working in the random-oracle and ideal-tweakable-cipher models instead. The random oracle [8] ideal functionality was already defined by Hofheinz and Müller-Quade in [32] (see Appendix A). Similarly, it is straightforward to derive the functionality for the ideal tweakable cipher primitive [37], see Appendix A. Note that for both, since the session identifier *sid* will be included either in the input for the random oracle, or in the tweak for the ideal-tweakable cipher, we can have one instantiation of each only, in the joint state, and not different instantiations for each session, which is more realistic. Note however that in some cases (in the first flow, see Figure 3), only some part of the session identifier will be included, which could lead to collisions with other sessions sharing this part of the *sid* (which by construction of the protocol are the sessions generated by the split functionality, see below). It could lead to a problem in the simulation, and more precisely in case of programming the random oracle or the ideal cipher. But programming is used in the proof only for honest players in a session. And with the split functionality, honest players are separated into various sets that make a *partition* (they are all disjoint): since a player cannot belong to two distinct sessions, this means that the simulator will have to program the random oracle or the decryption only once and

that no problem occurs (except, of course, if the adversary has already asked the critical query, e.g. an encryption leading to the particular ciphertext, but this happens with negligible probability only).

Split Functionalities. Without any strong authentication mechanisms, the adversary can always partition the players into disjoint subgroups and execute independent sessions of the protocol with each subgroup, playing the role of the other players. Such an attack is unavoidable since players cannot distinguish the case in which they interact with each other from the case where they interact with the adversary. The authors of [6] addressed this issue by proposing a new model based on *split functionalities* which guarantees that this attack is the only one available to the adversary.

Given a functionality $\mathcal{F}$, the split functionality $s\mathcal{F}$ proceeds as follows:
Initialization:

- Upon receiving $(\texttt{Init}, sid)$ from party P_i, send $(\texttt{Init}, sid, P_i)$ to the adversary.
- Upon receiving a message $(\texttt{Init}, sid, P_i, H, sid_H)$ from $\mathcal{A}$, where H is a set of party identities, check that P_i has already sent $(\texttt{Init}, sid)$ and that for all recorded $(H', sid_{H'})$, either $H = H'$ and $sid_H = sid_{H'}$ or H and H' are disjoint and $sid_H \neq sid_{H'}$. If so, record the pair (H, sid_H), send $(\texttt{Init}, sid, sid_H)$ to P_i, and invoke a new functionality $(\mathcal{F}, sid_H)$ denoted as $\mathcal{F}_H$ and with set of honest parties H.

Computation:

- Upon receiving $(\texttt{Input}, sid, m)$ from party P_i, find the set H such that $P_i \in H$ and forward m to $\mathcal{F}_H$.
- Upon receiving $(\texttt{Input}, sid, P_j, H, m)$ from $\mathcal{A}$, such that $P_j \notin H$, forward m to $\mathcal{F}_H$ as if coming from P_j.
- When $\mathcal{F}_H$ generates an output m for party $P_i \in H$, send m to P_i. If the output is for $P_j \notin H$ or for the adversary, send m to the adversary.

Fig. 1. Split Functionality $s\mathcal{F}$

The split functionality is a generic construction based upon an ideal functionality: Its description can be found on Figure 1. In the initialization stage, the adversary adaptively chooses disjoint subsets of the honest parties (with a unique session identifier that is fixed for the duration of the protocol). More precisely, the protocol starts with a session identifier sid. Then, the initialization stage generates some random values which, combined together and with sid, create the new session identifier sid', shared by all parties which have received the same values – that is, the parties of the disjoint subsets. The important point here is that the subsets create a *partition* of the players, thus forbidding communication among the subsets. During the computation, each subset H activates a separate instance of the functionality $\mathcal{F}$. All these functionality instances are independent: The executions of the protocol for each subset H can only be related in the way the adversary chooses the inputs of the players it controls. The parties $P_i \in H$ provide their own inputs and receive their own outputs (see the first item of "computation" in Figure 1), whereas the adversary plays the role of all the parties $P_j \notin H$ (see the second item).

The Group Password-Based Key Exchange Functionality with Mutual Authentication. In this section, we discuss the $\mathcal{F}_{GPAKE}$ functionality (see Figure 2). The multi-session extension of our functionality would be similar to the one proposed by Canetti and Rabin [27]. Our starting points are the group key exchange functionality described in [34] and the (two party) password-based key exchange functionality given in [25]. Our aim is to combine the two of them and to add mutual authentication and (t, n)-contributiveness. The new definition still remains very general: letting $t = 1$, we get back the case in which the adversary may manage to set the key when it controls at least a player, as in [25].

First, notice that the functionality is not in charge of providing the passwords to the participants. Rather we let the environment do this. As already pointed out in [25], such an approach allows to model, for example, the case where some users may use the same password for different protocols and, more generally, the case where passwords are chosen according to some arbitrary distribution (i.e. not

necessarily the uniform one). Moreover, notice that allowing the environment to choose the passwords guarantees forward secrecy, basically for free. More generally, this approach allows to preserve security[1] even in those situations where the password is used (by the same environment) for other purposes.

In the following we denote by n the number of players involved in a given execution of the protocol. The functionality starts with an initialization step during which it basically waits for each player to notify its interest in participating to the protocol. More precisely, we assume that every player starts a new session of the protocol with input (`NewSession`, sid, P_i, Pid, pw_i), where P_i is the identity of the player, pw_i is its password and Pid represents the set of (identities of) players with whom it intends to share a session key. Once all the players (sharing the same sid and Pid) have sent their notification message, $\mathcal{F}_{GPAKE}$ informs the adversary that it is ready to start a new session of the protocol.

In principle, after the initialization stage is over, all the players are ready to receive the session key. However the functionality waits for $\mathcal{S}$ to send an "ok" message before proceeding. This allows $\mathcal{S}$ to decide the exact moment when the key should be sent to the players and, in particular, it allows $\mathcal{S}$ to choose the exact moment when corruptions should occur (for instance $\mathcal{S}$ may decide to corrupt some party P_i before the key is sent but after P_i decided to participate to a given session of the protocol, see [34]).

Once the functionality receives a message (sid, Pid, `ok`, sk) from $\mathcal{S}$, it proceeds to the key generation phase. This is done as follows. If all players in Pid share the same password and less than t players are corrupted, the functionality chooses a key sk' uniformly and at random in the appropriate key space. If all players in Pid share the same password but t or more players are corrupted, then the functionality allows $\mathcal{S}$ to fully determine the key by letting $sk' = sk$. In all the remaining cases no key is established.
REMARK. For sake of simplicity, we chose to integrate the contributiveness in the UC-functionality. However, one could say that this is not necessary and that one could have kept the original functionality for group password-based key exchange, and then studying the contributiveness as an extra property. But then, two security proofs would be needed.

Our definition of the $\mathcal{F}_{GPAKE}$ functionality deals with corruptions of players in a way quite similar to that of $\mathcal{F}_{GPAKE}$ in [34], in the sense that if the adversary has corrupted some participants, it may determine the session key, but here only if there are enough corrupted players. Notice however that $\mathcal{S}$ is given such power only before the key is actually established. Once the key is set, corruptions allow the adversary to know the key but not to choose it. Following [25], a correct password test is captured by marking the corresponding record as `compromised`, and a failed attempt by marking it as `interrupted`. Records that are neither `compromised` nor `interrupted` are initially marked as `fresh`. Once a key is established, all records in Pid are marked as `complete`. Changing the fresh status of a record whenever a password test occurs or a (valid) key is established, is aimed at limiting the number of password tests to at most one per player. However, the `TestPwd` queries seem unnecessary in the context of split functionalities, since by splitting the whole group in subgroups of one player each, the adversary can already test one password per player. However, adding this query just allows $\mathcal{A}$ to test an additional password per player, which does not change significantly its power in practice. Besides, these queries are needed in the security analysis of our protocol, which we tried to keep as efficient as possible. Designing an efficient protocol while getting rid of theses queries is an interesting open problem.

In any case, after the key generation, the functionality informs the adversary about the result, meaning with this that the adversary is informed on whether a key was actually established or not. In particular, this means that the adversary is also informed on whether the players share the same password or not. At first glance this may seem like a dangerous information to provide to the adversary. We argue, however, that this is not the case in our setting. Indeed, being all the passwords chosen by the environment, such an information could be available to the adversary anyway. Moreover, it does

[1] By "preserved" here we mean that the probability of breaking the scheme is basically the same as the probability of guessing the password.

not seem critical to hide the status of the protocol (i.e. if it completed correctly or not), as in practice this information is often easily obtained by simply monitoring its execution (if the players suddenly stop their communications, there must have been some problem).

Finally the key is sent to the players according to the schedule chosen by $\mathcal{S}$. This is formally modeled by means of key delivery queries. We assume that, once $\mathcal{S}$ asks to deliver the key to a player, the key is sent immediately.

The functionality $\mathcal{F}_{GPAKE}$ is parameterized by a security parameter k and the parameter t of the contributiveness. It interacts with an adversary $\mathcal{S}$ and a set of parties $P_1, \ldots, P_n$ via the following queries:

- **Initialization.** Upon receiving $(\texttt{NewSession}, sid, P_i, Pid, pw_i)$ from player P_i for the first time, where Pid is a set of at least two distinct identities containing P_i, record (sid, P_i, Pid, pw_i), mark it `fresh`, and send (sid, P_i, Pid) to $\mathcal{S}$. Ignore any subsequent query $(\texttt{NewSession}, sid, P_j, Pid', pw_j)$ where $Pid' \neq Pid$.
  If there are already $|Pid| - 1$ recorded tuples (sid, P_j, Pid, pw_j) for players $P_j \in Pid \setminus \{P_i\}$, then record $(sid, Pid, \texttt{ready})$ and send it to $\mathcal{S}$.
- **Password tests.** Upon receiving a query $(\texttt{TestPwd}, sid, P_i, Pid, pw')$ from the adversary $\mathcal{S}$, if there exists a record of the form (sid, P_i, Pid, pw_i) which is `fresh`:
  - If $pw_i = pw'$, mark the record `compromised` and reply to $\mathcal{S}$ with "correct guess".
  - If $pw_i \neq pw'$, mark the record `interrupted` and reply to $\mathcal{S}$ with "wrong guess".
- **Key Generation.** Upon receiving a message $(sid, Pid, \texttt{ok}, sk)$ from $\mathcal{S}$ where there exists a recorded tuple $(sid, Pid, \texttt{ready})$, then, denote by n_c the number of corrupted players, and
  - If all $P_i \in Pid$ have the same passwords and $n_c < t$, choose $sk' \in \{0,1\}^k$ uniformly at random and store (sid, Pid, sk'). Next, for all $P_i \in Pid$ mark the record (sid, P_i, Pid, pw_i) `complete`.
  - If all $P_i \in Pid$ have the same passwords and $n_c \geq t$, store (sid, Pid, sk). Next, for all $P_i \in Pid$ mark the record (sid, P_i, Pid, pw_i) `complete`.
  - In any other case, store $(sid, Pid, \texttt{error})$. For all $P_i \in Pid$ mark the record (sid, P_i, Pid, pw_i) `error`.

  When the key is set, report the result (either `error` or `complete`) to $\mathcal{S}$.
- **Key Delivery.** Upon receiving a message $(\texttt{deliver}, \texttt{b}, sid, P_i)$ from $\mathcal{S}$, then if $P_i \in Pid$ and there is a recorded tuple (sid, Pid, α) where $\alpha \in \{0,1\}^k \cup \{\texttt{error}\}$, send (sid, Pid, α) to P_i if `b` equals `yes` or $(sid, Pid, \texttt{error})$ if `b` equals `no`.
- **Player Corruption.** If $\mathcal{S}$ corrupts $P_i \in Pid$ where there is a recorded tuple (sid, P_i, Pid, pw_i), then reveal pw_i to $\mathcal{S}$. If there also is a recorded tuple (sid, Pid, sk), that has not yet been sent to P_i, then send (sid, Pid, sk) to $\mathcal{S}$.

Fig. 2. Functionality $\mathcal{F}_{GPAKE}$

Notice that, the mutual authentication indeed means that if one of the players accepts, then all players share the key material; but, it doesn't mean that they all accept. Indeed, we cannot assume that all the flows are correctly forwarded by the adversary: it can modify just one flow, or at least omit to deliver one flow. This attack, called *denial of service*, is modeled in the functionality by the key delivery: the adversary can choose whether it wants the player to receive or not the good key/messages simply with the help of the keyword `b` set to `yes` or `no`.

3 Our Scheme

Description. Our solution builds on an earlier protocol by Abdalla *et al.* [2] and is described in Figure 3. Let $\mathcal{E}$ and $\mathcal{D}$ be the encryption and decryption schemes of an ideal tweakable cipher scheme. We denote by $\mathcal{E}_{pw}^{\ell}(m)$ an encryption of the message m using the ideal tweakable cipher, label ℓ and password pw. Similarly, the decryption is denoted as $\mathcal{D}_{pw}^{\ell}(c)$. The protocol uses five different random oracles, denoted by $\mathcal{H}_i$ for all $i = 0, \ldots, 4$. We denote by q_{h_i} the number of queries made to the oracle $\mathcal{H}_i$ ($q_h = q_{h_0} + \cdots + q_{h_4}$), and by $k_i = 2^{\ell_i}$ the output size: $\mathcal{H}_i \colon \{0,1\}^k \to \{0,1\}^{\ell_i}$. For an optimal instantiation, we will assume that for all i, $\ell_i = 2k$ (collisions), where k is the security parameter. Finally let $(\mathsf{SKG}, \mathsf{Sign}, \mathsf{Ver})$ be a one-time signature scheme, SKG being the signature key generation, Sign

the signing algorithm and Ver the verifying algorithm. Note that we do not require a *strong* one-time signature: Here, the adversary is allowed to query the signing oracle at most once, and should not be able to forge a signature of *another* authenticator.

Informally, and omitting the details, the algorithm can be described as follows: First, each player chooses a random exponent x_i and computes $z_i = g^{x_i}$ and an encryption z_i^* of z_i. It then applies SKG to generate a pair $(\mathrm{SK}_i, \mathrm{VK}_i)$ of signature keys, and commits to the values VK_i and z_i^*. In the second round, it reveals these values (the use of the commitment will be explained later in this section). We stress that the second round does not begin until all commitments have been received. At this point, the session identifier becomes $ssid' = ssid \| c_1 \| \ldots \| c_n$. It will be included, and verified, in all the subsequent hash values. Then, after verifying the commitments of the others, each couple (P_i, P_{i-1}) of players computes a common Diffie-Hellman value $Z_i = g^{x_i\, x_{i-1}}$, leading to a hash value X_i for each player. Each player then commits to this X_i, and once all these commitments have been received, it reveals the value X_i. In the next round, the players check this second round of commitments and compute an authenticator and the associated signature. Finally, the players check these authenticators and signatures, and if they are all correct, they compute the session key and mark their session as `complete`.

As soon as a value received by P_i doesn't match with the expected value, it aborts, setting the key $sk_i = \texttt{error}$. In particular, every player checks that $c_i = \mathcal{H}_3(ssid, z_i^*, \mathrm{VK}_i, i)$, $c_i' = \mathcal{H}_4(ssid', X_i, i)$ and $\mathsf{Ver}(\mathrm{VK}_i, \mathcal{A}uth_i, \sigma_i) = 1$.

We now highlight some of the differences between the scheme of [2] and our scheme, described in Figure 3. First, our construction does not require any random nonce in the first round (the session id constructed after the first flow is enough). Moreover, to properly implement the functionality, we return an error message to the players whenever they don't share the same password (mutual authentication). For sake of simplicity, two additional modifications are not directly related to the functionality. First, we use an ideal tweakable cipher rather than a different symmetric key for each player. Second, the values X_i's are here computed as the xor of two hashes.

Due to the split functionality, the players are partitioned according to the values they received during the first round (*i.e.* before the dotted line in Figure 3). All the c_i are shared among them – and thus the z_i^* and VK_i due to the random oracle $\mathcal{H}_3$ – and their session identifier becomes $ssid' = ssid \| c_1 \| \ldots \| c_n$. In round 3, the signature added to the authentication flow prevents the adversary from being able to change an authenticator to another value. At the beginning of each flow, the players wait until they have received all the other values of the previous flow before sending their new one. This is particularly important between `flow(1a)` and `flow(1b)` and similarly between `flow(2a)` and `flow(2b)`. Since the session identifier $ssid'$ is included in all the hash values, and in the latter signature, only players in the same subset can accept and conclude with a common key.

Finally, the contributory property is ensured by the following modification: In the first and second rounds, each player starts by sending a commitment of the value it has just computed (using a random oracle), denoted as c_i and c_i'. Due to this commitment, it is impossible for a player to compute its z_i^* (or X_i) once it has seen the others: Every player has to commit its z_i^* (or X_i) at the same time as the others, and this value cannot depend on the other values sent by the players.

Finally we point out that, in our proof of security, we don't need to assume that the players erase any ephemeral value before the end of the computation of the session key.

Computational Diffie-Hellman Assumption. Denote by $G = \langle g \rangle$ a finite cyclic (multiplicative) group of prime order q. If x, y are chosen uniformly at random in $\mathbb{Z}_q^*$, the CDH assumption states it is computationally intractable to output g^{xy} given g, g^x and g^y.

It is easy to see that if P_i and P_{i+1} have the same passwords, they will compute in round 2 the same $Z_{i+1} = (z_{i+1})^{x_i} = g^{x_i x_{i+1}} = (z_i)^{x_{i+1}}$. If the passwords are different, we denote by Z_i^R a value computed by P_i on its right side, and Z_{i+1}^L a value computed by P_{i+1} on its left side. We have:

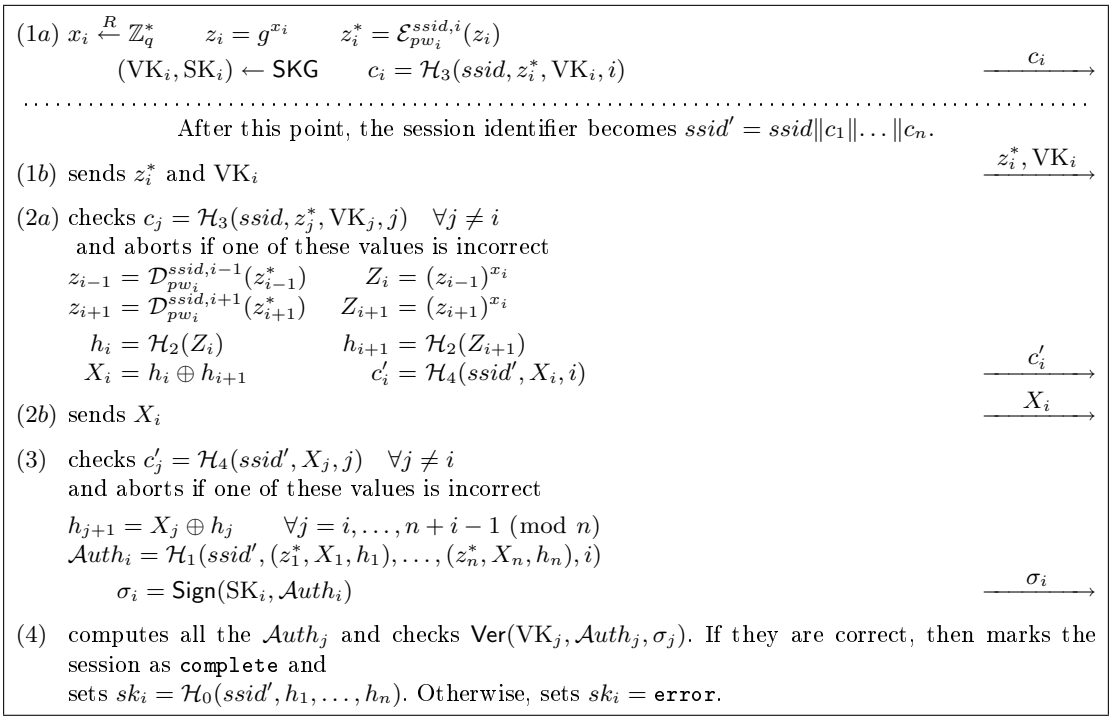

Fig. 3. Description of the protocol for player P_i, with index i and password pw_i

$$Z_i^R = CDH_g(\mathcal{D}_{pw_i}^{ssid,i-1}(z_{i-1}^*), \mathcal{D}_{pw_i}^{ssid,i}(z_i^*)) \qquad h_i^R = \mathcal{H}_2(Z_i^R)$$
$$Z_{i+1}^L = CDH_g(\mathcal{D}_{pw_i}^{ssid,i}(z_i^*), \mathcal{D}_{pw_i}^{ssid,i+1}(z_{i+1}^*)) \qquad h_{i+1}^L = \mathcal{H}_2(Z_{i+1}^L)$$

(here CDH_g denotes the Diffie-Hellman function in base g that given on input g^a, g^b outputs g^{ab}), and then Z_{i+1}^L and Z_{i+1}^R are likely different.

Pictorially, the situation can be summarized as follows

$$P_{i-1} \xleftrightarrow{Z_i^L \neq Z_i^R} P_i \xleftrightarrow{Z_{i+1}^L \neq Z_{i+1}^R} P_{i+1}$$

Each P_i computes $X_i = h_i^R \oplus h_{i+1}^L$ and thus, once the values X_j's are published, all the players can iteratively compute all the h_j's required to compute the authenticators $\mathcal{A}uth_i$ and later the session key sk_i.

Our Main Theorem. Let $\widehat{s\mathcal{F}_{GPAKE}}$ be the multi-session extension of the split functionality $s\mathcal{F}_{GPAKE}$ and let $\mathcal{F}_{RO}$ and $\mathcal{F}_{ITC}$ be the ideal functionalities that provide a random oracle and an ideal tweakable cipher to all parties.

Theorem 1 *The protocol presented in Figure 3 securely realizes $\widehat{s\mathcal{F}_{GPAKE}}$ in the $(\mathcal{F}_{RO}, \mathcal{F}_{ITC})$-hybrid model, in the presence of adaptive adversaries, and is $(n/2, n)$-contributory.*

Corollary 1. *This protocol, along with small modifications described in Appendix B.2 (with $n/2$ parallel executions of the original scheme), securely realizes $\widehat{s\mathcal{F}_{GPAKE}}$ in the $(\mathcal{F}_{RO}, \mathcal{F}_{ITC})$-hybrid model, in the presence of adaptive adversaries, and is $(n-1, n)$-contributory.*

Note that if the signature scheme is subliminal channel free [40] (*e.g.* only one signature σ is valid for a given pair $(\mathrm{VK}, \mathcal{A}uth)$), then after the first round (1a), everything is deterministic, and thus no subliminal channel is available for an adversary to leak any information to an eavesdropper.

4 Proof of Theorem 1

We prove the protocol to be $(n/2, n)$-contributory, and show in Appendix B.2 how to get the $(n-1, n)$-contributiveness, with some parallel executions. Note that $n/2$ here implicitly means $\lfloor n/2 \rfloor$ when n is odd.

We start by giving an attack showing that the $(n/2+1)$-contributory cannot be satisfied. For sake of simplicity, assume that n is odd and consider the following situation in which there are $\lfloor n/2 \rfloor$ honest players (denoted as P_i) and $\lfloor n/2 \rfloor + 1$ corrupted players (denoted as $\mathcal{A}_i$, since they are under the control of the adversary $\mathcal{A}$):

$$\mathcal{A}_1 \quad P_1 \quad \mathcal{A}_2 \quad P_2 \quad \mathcal{A}_3 \ldots P_{\lfloor n/2 \rfloor} \quad \mathcal{A}_{\lfloor n/2 \rfloor} \quad \mathcal{A}_{\lfloor n/2 \rfloor + 1}$$

Since $\mathcal{A}$ knows the random values of the corrupted players, it learns, from flows $(1b)$, the values Z_i, and thus $h_i = \mathcal{H}_2(Z_i)$ for $i = 1, \ldots, n-1$, before it plays on behalf of $\mathcal{A}_{\lfloor n/2 \rfloor + 1}$. Even if it cannot modify anymore z_n in flow $(1b)$ (already committed to in c_n), it can choose Z_n to bias the value h_n, and thus the final key, that is defined as $sk = \mathcal{H}_0(ssid', h_1, \ldots, h_n)$: this is introduced in flow $(2a)$. Such an inconsistent value is possible since no honest player can verify the value of Z_n: this is the Diffie-Hellman value between two corrupted players.

More generally, if $\mathcal{A}$ controls enough players so that each honest player is between two corrupted players, then it can learn, from flows $(1b)$, the values X_i that will be sent in flows $(2b)$. If two corrupted players are neighbors, they can send a value X_i of their choice, since it comes from a Diffie-Hellman value between these two corrupted players. In the attack above, the adversary could learn all the h_i early enough, so that its control on h_n could bias the key. If it can control an h_i, without knowing the other values, there is still enough entropy in the key derivation: the final key is uniformly distributed: contributiveness.

We now prove the $(n/2, n)$-contributiveness, using the above intuition. We need to construct, for any real-world adversary $\mathcal{A}$ (interacting with real parties running the protocol), an ideal-world adversary $\mathcal{S}$ (interacting with dummy parties and the functionality $\widehat{s\mathcal{F}_{GPAKE}}$) such that no environment $\mathcal{Z}$ can distinguish between an execution with $\mathcal{A}$ in the real world and $\mathcal{S}$ in the ideal world with non-negligible probability.

We incrementally define a sequence of games starting from the one describing a real execution of the protocol in the real world, and ending up with game $\mathbf{G_8}$ which we prove to be indistinguishable with respect to the ideal experiment. The key point will be $\mathbf{G_7}$. $\mathbf{G_0}$ is the real-world game. In $\mathbf{G_1}$, we start by simulating the encryption, decryption and hash queries, canceling some unlikely events (such as collisions). Granted the ideal tweakable cipher model (see details in Appendix C), we can extract the passwords used by $\mathcal{A}$ for players corrupted from the beginning of the session. $\mathbf{G_2}$ and $\mathbf{G_3}$ allow $\mathcal{S}$ to be sure that the authenticators for non-corrupted players are always oracle-generated. In $\mathbf{G_4}$, we show how to deal with the simulation of the first flows. In $\mathbf{G_5}$, we deal with only oracle-generated flows. In $\mathbf{G_6}$, we deal with (possibly) non-oracle-generated flows from round 2. $\mathbf{G_7}$ is the crucial game, where we show how to simulate the non-corrupted players without the knowledge of their passwords, even in the case of corruptions before round 2. Finally, we show that $\mathbf{G_8}$, in which we only replace the hybrid queries by the real ones, is indistinguishable from the ideal game.

To this aim, we first describe three hybrid queries that are going to be used in the games. The `GoodPwd` query checks whether the password of some player is the one we have in mind or not. The `SamePwd` query checks if the players share the same password, without disclosing it. The `Delivery` query provides the player with the session key. In some games, the simulator has actually access to the honest players, and thus to their passwords (and always knows the passwords committed by the adversary for corrupted users granted the ideal tweakable cipher). In such a case, these queries can be easily implemented by letting $\mathcal{S}$ look at these passwords. When the players are entirely simulated, $\mathcal{S}$ will replace the queries above with `TestPwd`, `Key Generation` and `Key Delivery` queries to the ideal functionality.

Following [25], we say that a flow is *oracle-generated* if it was sent by an honest player (our simulation) and arrives without any alteration to the player it was meant to. We say it is *non-oracle-generated* otherwise, that is either if it was sent by an honest player and modified by the adversary, or if it was sent by a corrupted player or a player impersonated by the adversary: in all these cases, we say that the sender is an *attacked* player. In brief, our simulation controls the random coins of oracle-generated flows, whereas the adversary may control them in non-oracle-generated flows.

Note that since we consider the split functionality, the players have been partitioned in sets according to what they received during the very first flow `flow(1a)`. In the following, we can thus assume that all the players have received the same `flow(1a)` and `flow(1b)` (under the binding property of the commitment $\mathcal{H}_3$). Oracle-generated flows `flow(1a)` have been sent by players that will be considered honest in this session, whereas non-oracle-generated flows have been sent by the adversary, the corresponding players are thus assumed corrupted from the beginning of the session, since the adversary has chosen the password. Note that an advantage of this model (using both a random oracle for the commitment, and an ideal tweakable cipher for the encryption) is that we know the passwords used by the adversary (and thus corrupted players) in this first round, by simply looking in the tables that will be defined in $\mathbf{G_2}$, and they are the same in the view of any honest player. The extraction of the password may fail, if the adversary has computed either the ciphertext or the commitment at random, but then it has no chance to make the protocol conclude successfully. Also note that if `flow(2a)` is oracle-generated, then `flow(2b)` must be oracle-generated also with overwhelming probability, due to $\mathcal{H}_4$, as above. As a result, we set $sk_i = \texttt{error}$ whenever an inconsistency is noted by a player (incorrect commitment opening, or invalid signature). The latter then aborts its execution.

Adaptive Corruptions and Connected Components. For simplicity, we consider that the simulator maintains, for each honest player, a list symbolizing its internal state:

$$\Lambda_i = (pw_i, SK_i, x_i, z_i, z_i^*, c_i, Z_i^R, Z_{i+1}^L, h_i^R, h_{i+1}^L, X_i, c_i'),$$

where the superscripts L and R denote the neighbor with whom the value is shared (the left or the right neighbor). When a player gets corrupted, the simulator has to provide the adversary with this list. Most of the fields will be chosen at random during the simulation, with the remaining values initially set to $\perp$. As soon as a player gets corrupted, the simulator recovers its password, which will help to fill the other fields in a consistent way, granted the programmability of the random oracle and the ideal tweakable cipher.

The knowledge of P_i's password indeed helps the simulator to fill in the internal state (see below). Note that this allows $\mathcal{S}$ to send the values (in particular the X_i) in a way that remains consistent with respect to the view of the adversary, and even the environment. Informally, $\mathcal{S}$ does this by partitioning the set of players into a number of connected components. Each component consists of all the connected players sharing the same password (the one used to generate the first flow). Below, we show that all $\mathcal{S}$ has to do for the simulation to work is to make sure the produced values are consistent only for the players belonging to the same components. Indeed, for neighbor players belonging to different components, $\mathcal{S}$ can basically produce completely unrelated values, without worrying about being caught, since the decrypted values z_i are unrelated, from the beginning of the protocol.

Simulator: Session Initialization. The aim of the first flow is to create the subsets H of players involved in the same protocol execution (see the split functionality, Section 2 and Figure 1). $\mathcal{S}$ chooses the values $(\mathrm{SK}_i, \mathrm{VK}_i)$ on behalf of the honest players and the value z_i^* at random, rather than asking an encryption query (it does not know the passwords of the players), then computes and sends the commitments c_i to $\mathcal{A}$. The environment initializes a session for each honest (dummy) player, which is modeled by the `Init` queries sent to the split functionality. The adversary (from the view of the c_i of the honest players) makes its decision about the subgroups it wants to make: it sends c_i on behalf of the players it wants to impersonate (they will become corrupted from the beginning of the session). We then define the H sets according to the received $\{c_j\}$: the honest players that have received the

same $\{c_j\}$ (possibly modified by the adversary) are in the same subgroup H. The simulator forwards these sets H (which make a partition of all the honest players) to the split functionality. The latter then initializes ideal functionalities with sid_H, for each subgroup H: all the players in the same session received and thus use the same $\{c_j\}$. The environment gets back this split, via the dummy players, and then sends the `NewSession` queries on behalf of the latter, according to the appropriate sid_H. The simulator uses the commitments sent from the adversary to extract the password used (granted the ideal tweakable cipher), and thus sends the appropriate `NewSession` queries on behalf of the corrupted players (note that in case that no password can be extracted, a random one is used). Then, we can focus on a specific session $ssid' = sid_H$ for some set H.

Simulator: Main Idea. In a nutshell, the simulation of the remaining of the protocol depends on the knowledge of the passwords by the simulator. First, if the simulator knows the password of a player, it does everything honestly for every player belonging to its connected component. Otherwise, it sets everything at random. In case of corruption, $\mathcal{S}$ learns the password, and can program the oracles (random oracle and ideal tweakable cipher) and fill in the internal state of the player (and of all the players in its connected component) in a consistent way. This last phase, which consists in programming the oracle, may fail, but only if the adversary can solve an intractable problem (computational Diffie-Hellman).

More precisely, in most of the cases, the simulator $\mathcal{S}$ just follows the protocol on behalf of all the honest players. The main difference between the simulated players and the real honest players is that $\mathcal{S}$ does not engage on a particular password on their behalf. However, if $\mathcal{A}$ generates/modifies a `flow(1a)` or a `flow(2a)` message that is delivered to player P in session $ssid'$, then $\mathcal{S}$ extracts the password pw, granted the ideal tweakable cipher, and uses it in a `TestPwd` query to the functionality. If this is a correct guess, then $\mathcal{S}$ uses this password on behalf of P, and proceeds with the simulation.

The key point of the simulation consists in sending coherent X_i's, whose values completely determine the session key. To this aim, we consider two cases. First, if the players are all honest and share the same password, the h_i's are chosen at random, but identically between two neighbors. If they do not share the same passwords, they are simply set at random. Second, if there are corrupted players among the players, the simulator determines the connected components, as described above, and the trick consists in making the simulation coherent within those components since the adversary has no means of guessing what happens between two components (the passwords are different).

If a session aborts or terminates, $\mathcal{S}$ reports it to $\mathcal{A}$. If the session terminates with a session key sk, then $\mathcal{S}$ makes a `Key Delivery` call to $\widehat{\mathcal{F}}_{GPAKE}$, specifying the session key. But recall that unless enough players are corrupted, $\widehat{\mathcal{F}}_{GPAKE}$ will ignore the key specified by $\mathcal{S}$, and thus we do not have to bother with the key in these cases.

5 Conclusion

This paper investigates a stronger security notion against insider adversaries, for password-based group AKE. The protocol as presented in Section 3 achieves $(n/2, n)$-contributiveness; We also show how to achieve $(n-1, n)$-contributiveness in Appendix B.2.

Acknowledgments

This work was supported in part by the French ANR-07-SESU-008-01 PAMPA Project and the European Commission through the IST Program under Contract ICT-2007-216646 ECRYPT II.

References

1. M. Abdalla, J.-M. Bohli, M. I. Gonzalez Vasco, and R. Steinwandt. (Password) authenticated key establishment: From 2-party to group. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 499–514. Springer, Feb. 2007.

2. M. Abdalla, E. Bresson, O. Chevassut, and D. Pointcheval. Password-based group key exchange in a constant number of rounds. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 427–442. Springer, Apr. 2006.
3. M. Abdalla, D. Catalano, C. Chevalier, and D. Pointcheval. Efficient two-party password-based key exchange protocols in the UC framework. In T. Malkin, editor, *CT-RSA 2008*, LNCS. Springer, Apr. 2008.
4. M. Abdalla and D. Pointcheval. A scalable password-based group key exchange protocol in the standard model. In X. Lai and K. Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 332–347. Springer, Dec. 2006.
5. G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *ACM CCS 98*, pages 17–26. ACM Press, Nov. 1998.
6. B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure computation without authentication. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 361–377. Springer, Aug. 2005.
7. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, May 2000.
8. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.
9. M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Aug. 1994.
10. M. Bellare and P. Rogaway. Provably secure session key distribution: The three party case. In *27th ACM STOC*, pages 57–66. ACM Press, May / June 1995.
11. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.
12. J.-M. Bohli, M. I. Gonzalez Vasco, and R. Steinwandt. Password-authenticated constant-round group key establishment with a common reference string. Cryptology ePrint Archive, Report 2006/214, 2006. `http://eprint.iacr.org/`.
13. J.-M. Bohli, M. I. G. Vasco, and R. Steinwandt. Secure group key establishment revisited. *Int. J. Inf. Secur.*, 6(4):243–254, 2007.
14. V. Boyko, P. D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. Springer, May 2000.
15. E. Bresson, O. Chevassut, and D. Pointcheval. Provably authenticated group Diffie-Hellman key exchange – the dynamic case. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 290–309. Springer, Dec. 2001.
16. E. Bresson, O. Chevassut, and D. Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 321–336. Springer, Apr. / May 2002.
17. E. Bresson, O. Chevassut, and D. Pointcheval. Group Diffie-Hellman key exchange secure against dictionary attacks. In Y. Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 497–514. Springer, Dec. 2002.
18. E. Bresson, O. Chevassut, and D. Pointcheval. Security proofs for an efficient password-based key exchange. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM CCS 03*, pages 241–250. ACM Press, Oct. 2003.
19. E. Bresson, O. Chevassut, and D. Pointcheval. A security solution for IEEE 802.11's ad-hoc mode: Password authentication and group Diffie-Hellman key exchange. *International Journal of Wireless and Mobile Computing*, 2(1):4–13, 2007.
20. E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater. Provably authenticated group Diffie-Hellman key exchange. In *ACM CCS 01*, pages 255–264. ACM Press, Nov. 2001.
21. E. Bresson and M. Manulis. Securing group key exchange against strong corruptions and key registration attacks. *Int. J. Appl. Cryptol.*, 1(2):91–107, 2008.
22. M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system (extended abstract). In A. D. Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 275–286. Springer, May 1994.
23. M. Burmester and Y. Desmedt. A secure and scalable group key exchange system. *Information Processing Letters*, 94(3):137–143, May 2005.
24. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
25. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, May 2005.
26. R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474. Springer, May 2001.
27. R. Canetti and T. Rabin. Universal composition with joint state. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Aug. 2003.
28. Y. Desmedt, J. Pieprzyk, R. Steinfeld, and H. Wang. A non-malleable group key exchange protocol robust against active insiders. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *ISC 2006*, volume 4176 of *LNCS*, pages 459–475. Springer, Aug. / Sept. 2006.
29. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

30. R. Dutta and R. Barua. Password-based encrypted group key agreement. *International Journal of Network Security*, 3(1):30–41, July 2006. http://isrc.nchu.edu.tw/ijns.
31. R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. In E. Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 524–543. Springer, May 2003. http://eprint.iacr.org/2003/032.ps.gz.
32. D. Hofheinz and J. Müller-Quade. Universally composable commitments using random oracles. In M. Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, Feb. 2004.
33. J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 475–494. Springer, May 2001.
34. J. Katz and J. S. Shin. Modeling insider attacks on group key-exchange protocols. In V. Atluri, C. Meadows, and A. Juels, editors, *ACM CCS 05*, pages 180–189. ACM Press, Nov. 2005.
35. N. Koblitz and A. Menezes. Another look at "provable security". II. (invited talk). In R. Barua and T. Lange, editors, *INDOCRYPT 2006*, volume 4329 of *LNCS*, pages 148–175. Springer, Dec. 2006.
36. H. Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 546–566. Springer, Aug. 2005.
37. M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 31–46. Springer, Aug. 2002.
38. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the Association for Computing Machinery*, 21(21):993–999, Dec. 1978.
39. V. Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999.
40. G. J. Simmons. The prisoners' problem and the subliminal channel. In D. Chaum, editor, *CRYPTO'83*, pages 51–67. Plenum Press, New York, USA, 1984.

A The Ideal Functionalities

The Random Oracle Functionality. The random oracle ideal functionality has already been defined by Hofheinz and Müller-Quade in [32]. We recall it in Figure 4.

Functionality $\mathcal{F}_{RO}$

The functionality $\mathcal{F}_{RO}$ proceeds as follows, running on security parameter k, with parties $P_1,\ldots,P_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{RO}$ keeps a list L (which is initially empty) of pairs of bitstrings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from some party P_i or from $\mathcal{S}$, do:
  - If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^k$ in the list L, set $h := \tilde{h}$.
  - If there is no such pair, choose uniformly $h \in \{0,1\}^k$ and store the pair (m, h) in L.

  Once h is set, reply to the activating machine (i.e., either P_i or $\mathcal{S}$) with (sid, h).

Fig. 4. Functionality $\mathcal{F}_{RO}$

The Ideal Tweakable Cipher Functionality. Very informally, a tweakable cipher is a block cipher that allows for a second input, called the tweak, on top of the key and the usual plaintext or ciphertext input. The tweak is essentially a label that, along with the key, selects the permutation computed by the cipher. The basic idea is that if changing tweaks is sufficiently lightweight (with respect to a key setup operation), then some interesting new operation modes become possible.

As for random oracles, it is straight-forward to derive the ideal functionality for the ideal tweakable cipher primitive. The functionality $\mathcal{F}_{ITC}$ takes as input the security parameter k, and keeps a (initially empty) list L containing 4−tuples of bitstrings and a number of (initially empty) sets $C_{key,label}$ and $M_{key,label}$ defined by

$$C_{key,label} = \{c \mid \exists m \ (key, label, m, c) \in L\}$$
$$M_{key,label} = \{m \mid \exists c \ (key, label, m, c) \in L\}.$$

It interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $P_1,\ldots,P_n$ by means of several queries. For more details, please refer to Figure 5.

Functionality $\mathcal{F}_{ITC}$

The functionality $\mathcal{F}_{ITC}$ takes as input the security parameter k, and interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $P_1,\ldots,P_n$ by means of the following queries:

- $\mathcal{F}_{ITC}$ keeps a (initially empty) list L containing 4−tuples of bitstrings and a number of sets $C_{key,label}, M_{key,label}$ (initially empty).
- **Upon receiving a query $(sid, ENC, key, label, m)$ (with $m \in \{0,1\}^k$) from some P_i or $\mathcal{S}$:**
  - If there is a 4−tuple $(key, label, m, \tilde{c})$ for some $\tilde{c} \in \{0,1\}^k$ in the list L, set $c := \tilde{c}$.
  - If there is no such record, choose uniformly c in $\{0,1\}^k \setminus C_{key,label}$ which is the set consisting of ciphertexts not already used with key and $label$. Next, it stores the 4−tuple $(key, label, m, c)$ in the list L.

  Once c is set, reply to the activating machine with (sid, c).
- **Upon receiving a query $(sid, DEC, key, label, c)$ (with $c \in \{0,1\}^k$) from some party P_i or $\mathcal{S}$:**
  - If there is a 4−tuple $(key, label, \tilde{m}, c)$ for some $\tilde{m} \in \{0,1\}^k$ in the list L, set $m := \tilde{m}$.
  - If there is no such record, choose uniformly m in $\{0,1\}^k \setminus M_{key,label}$ which is the set consisting of plaintexts not already used with key and $label$. Next, it stores the 4−tuple $(key, label, m, c)$ in the list L.

  Once m is set, reply to the activating machine with (sid, m).

Fig. 5. Functionality $\mathcal{F}_{ITC}$

B Generalization

B.1 Means of Authentication

When defining the ideal functionality for password-based group AKE in Section 2, we require all users to have the same password in order to establish a common secret key. There are, however, other ways of defining the ideal functionality which may be better suited to a particular application at hand. For example, consider the case in which each pair of users has a different password associated with it. In this scenario, we could envisage a definition which allows users to establish a common secret as long as each user in the group shares a common password with its neighbors (assuming a particular ordering of the users). We opted for the first formulation in this paper due to its simplicity.

As in [6], our results can also be extended to the case of partially authenticated networks, where some of the parties involved in the protocol may have authenticated links available to them. As noted by Barak *et al.* [6], this seems to be a much more realistic setting than the standard one that assumes that either all pairs of parties or none of them are connected via authenticated channels.

B.2 $(n-1, n)$ Contributiveness

We have just shown that if there are two non-corrupted neighbors at some place in the cycle (see $\boldsymbol{G}_{7a}$), then the key is completely unpredictable to the adversary, which is exactly $(n/2, n)$-contributiveness. We now show how to extend this result to $(n-1, n)$-contributiveness.

Note that one can construct $p = \binom{n}{2}$ rings such that all pairs of players will be neighbors in one of them. To achieve this result, one simply considers all possible pairs of players and then complete the rings in an arbitrary way. It is actually possible to obtain a linear number of rings (see Lemma 2)

We now prove how such a construction can help the protocol achieve the $(n-1, n)$-contributiveness. Consider a protocol consisting of p parallel executions of the protocol, in which the $ssid$ includes the number of the execution. To avoid malleability attacks in which the adversary replays messages in different orders within the same session, we also ask players not to send separate values of c_i and c'_i for each execution, but rather send unique values of c_i and c'_i for all executions. Similarly, a unique signing and verification key is enough: a player signs an authenticator that involves the key material of all the parallel executions. At the end of these parallel executions, each player has obtained p keys. If one of them is an error message, it is given an error message. Otherwise, the final session key is defined as the hash of all these keys.

Assume that the players all have the same password. Then, if there are at least two non-corrupted players, we consider the execution in which they are neighbors. We showed in $\boldsymbol{G}_7$ that, in this case, the key of this execution is unpredictable to the adversary. This leads to a final key which is unpredictable to the adversary, and thus to the $(n-1,n)$-contributiveness.

Lemma 2. *We can construct $\lceil n/2 \rceil$ rings of the n players such that for every pair (P_i, P_j) of players, P_i and P_j are neighbors in at least one ring.*

Proof. One can easily verify this result for $n = 2$ and 3, and we prove it recursively. Consider an odd number n and assume that the result is true for $n-1$ players. Denote by $\mathcal{A}_{n-1}$ the algorithm for $n-1$ players. We prove that the result is true for n and then for $n+1$.

We first apply $\mathcal{A}_{n-1}$ for the players $P_1, \ldots, P_{n-1}$ and we insert in each ring the player P_n between two neighbors it has never seen until possible. We now prove that this manipulation will be possible until the $i_0 = \lfloor n/2 \rfloor^{th}$ ring.

Mark "s" a player which has already been a neighbor of P_n and "n" a player who has never been. We want to show that there always exist two neighbors of type "n" until ring i_0. The worst situation is "(sn)(sn)...(sn)nn...nn", where the seen players are always between two non-seen players, thus providing P_n from going into this place in the next ring.

Let i be a ring in which we could find a place for P_n between two "n" players. Since P_n sees two more different players in each ring, it has seen $2(i-1)$ players until ring i. The number of free places (between two "n") thus equals to (n-1) $- [2(i-1)] = n + 1 - 2i$. To be able to insert P_n, we need at least one free place, so that $n + 1 - 2i \geq 1$ and $i \leq n/2$.

We do this manipulation from rings 1 to $(n-1)/2$. Then, P_n has seen $n-1$ players as neighbors, that is, everyone. And $\mathcal{A}_{n-1}$ is finished, too, since it consists of $(n-1)/2$ rings. Now, the only players which have not met are those which should have met if we had not inserted P_n between them. But the pairs separated by P_n are all distinct by construction, so that we are able to make them all meet in only one ring (we simply juxtapose them). Finally, we found an algorithm for n players in $(n-1)/2+1 = \lceil n/2 \rceil$ rings.

We now show that the result is also true for $n+1$ under the same assumptions. As before, we use $\mathcal{A}_n$ and we insert P_{n+1} between two never seen players until possible. Similarly, the last possible ring is $(n+1)/2$, but we choose to stop at ring $(n-1)/2$. At this stage, P_{n+1} has seen $n-1$ players as neighbors, that is, everybody except one. But the last ring of $\mathcal{A}_n$ consists of a juxtaposition of independent pairs. Thus, we can insert P_{n+1} next to the last neighbor, without changing anything to the correctness of $\mathcal{A}_n$. Finally, if n is odd, we showed how to obtain $\mathcal{A}_n$ and $\mathcal{A}_{n+1}$ given $\mathcal{A}_{n-1}$.

C Further Details on the Proof

C.1 Description of the Games

Game $\boldsymbol{G}_0$: Game $\boldsymbol{G}_0$ is the real game in the random-oracle and ideal-tweakable-cipher models: the simulator simulates all the honest players as they would do according to the protocol, using the passwords sent by the environment.

Game $\boldsymbol{G}_1$: We modify the previous game by simulating the hash and the encryption/decryption oracles, in a quite natural and usual way. For the ideal tweakable cipher, we allow $\mathcal{S}$ to maintain a list Λ_ε of entries (queries, responses) of length $q_\varepsilon + q_\mathcal{D}$, composed of the two sub-lists: $\{(pw, (ssid, i), Y, \alpha, \mathcal{E}, Y^*)\}$ for the encryption queries, and $\{(pw, (ssid, i), Y, \alpha, \mathcal{D}, Y^*)\}$ (for decryption). The first (resp. second) sub-list is indeed used to indicate that the element Y (respectively Y^*) has been encrypted ("$\mathcal{E}$") (respectively decrypted ("$\mathcal{D}$")) to produce the ciphertext Y^* (resp. the plaintext Y) via a symmetric encryption algorithm that uses the key pw and tweak $(ssid, i)$. Actually, for a new encryption query, Y^* is randomly chosen, and α is set to $\perp$. For a new decryption query, α is randomly chosen in $\mathbb{Z}_q^*$,

and Y is set to g^α. Such a list is used by $\mathcal{S}$ to be able to provide answers which are consistent with the following requirements: 1) the same question for the same password/tweak pair will receive the same answer; 2) the simulated scheme (for each password/tweak) is actually a permutation over G; 3) in order to help $\mathcal{S}$ to later extract the password used in the first flow, there should not be two encryption entries of the form (question, answer) with identical ciphertext and tweak, but different passwords: a ciphertext (obtained by encryption) should correspond to a unique password. On the other hand, any decryption of a ciphertext not obtained by encryption will provide the discrete logarithm at the same time. $\mathcal{S}$ also maintains a list $\Lambda_{\mathcal{H}}$ of tuples (i, q, r) where $\mathcal{H}_i(q) = r$, used to properly manage the queries for the random oracles $\mathcal{H}_i$, excluding the collisions too. More precisely, we cancel the games where these two kinds of collisions appear.

More precisely, the list Λ_ε described above is actually composed of the two following sublists: $\{(pw, (ssid, i), Y, \alpha, \mathcal{E}, Y^*)\}$ and $\{(pw, (ssid, i), Y, \alpha, \mathcal{D}, Y^*)\}$. The first (resp. second) sublist is used to indicate that the element Y (respectively Y^*) has been encrypted ("$\mathcal{E}$") (respectively decrypted ("$\mathcal{D}$")) to produce the ciphertext Y^* (resp. Y) via a symmetric encryption algorithm that uses the key pw for user i in session $ssid$ (with tweak $(ssid, i)$). The role of α will be explained later on. The simulator manages the list of encryption and decryption queries through the following rules:

- For an encryption query $\mathcal{E}_{pw}^{ssid,i}(Y)$ such that $(pw, (ssid, i), Y, *, *, Y^*)$ appears in the list Λ_ε, the answer is Y^*. Otherwise, choose a random $Y^* \in G^*$. If $(*, (ssid, i), *, *, *, Y^*)$ already belongs to Λ_ε, then abort, else add the new record $(pw, (ssid, i), Y, \perp, \mathcal{E}, Y^*)$ to the list.
- For a decryption query $\mathcal{D}_{pw}^{ssid,i}(Y^*)$ such that $(pw, (ssid, i), Y, *, *, Y^*)$ appears in the list Λ_ε, the answer is Y. Otherwise, choose a random $Y \in G^*$. If $(*, (ssid, i), Y, *, *, Y^*)$ already belongs to Λ_ε, then abort, else add the new record $(pw, (ssid, i), Y, \perp, \mathcal{D}, Y^*)$ to the list.

The simulator also updates $\Lambda_{\mathcal{H}}$ for the hash queries using the following general rule (where n stands for $0, \ldots, 4$):

- For a hash query $\mathcal{H}_n(q)$ such that (n, q, r) appears in the list $\Lambda_{\mathcal{H}}$, the answer is r. Otherwise, choose a random $r \in \{0,1\}^{\ell_n}$. If $(n, *, r)$ already belongs to $\Lambda_{\mathcal{H}}$, abort, else add (n, q, r) to the list.

The birthday paradox bound implies that $\boldsymbol{G}_1$ and $\boldsymbol{G}_0$ are statistically indistinguishable for properly chosen oracle-output length.

Game $\boldsymbol{G}_2$: In this game, we reject an authenticator that is sent without having been asked to the oracle $\mathcal{H}_1$. It makes a difference if one rejects such an authenticator that is actually valid. But this, of course, happens only with negligible probability: $\boldsymbol{G}_2$ and $\boldsymbol{G}_1$ are statistically indistinguishable.

Game $\boldsymbol{G}_3$: We now furthermore reject any signed-authenticator that is non-oracle-generated whereas the player is still honest: the adversary does not know the signing key, and thus cannot correctly sign the authenticator. One can easily show that a difference between $\boldsymbol{G}_2$ and $\boldsymbol{G}_3$ would lead to an attack against the one-time signature scheme.

Game $\boldsymbol{G}_4$: In this game, we formally modify the way we simulate the honest players. The simulator still knows their passwords. In round 1, $\mathcal{S}$ sends a random value c_i on behalf of P_i and afterwards, when all c_j have been sent, it sends a random value z_i^* (chosen without asking the encryption oracle, since we try to avoid the use of the password), and the verification key VK_i, generated honestly. Finally, $\mathcal{S}$ sets $c_i = \mathcal{H}_3(ssid, z_i^*, \mathrm{VK}_i, i)$ by programming the oracle. There is a negligible risk that the simulation fails, if the adversary has already asked this query to the oracle, which probability is bounded by q_{h_3}/q. In round 2, $\mathcal{S}$ proceeds by using the password pw_i: it asks for the three decryption queries, on z_i^*, z_{i-1}^*, and z_{i+1}^*, with key pw_i and appropriate tweaks. Granted the way we simulate the decryption oracle, we learn z_{i-1}, z_i, and z_{i+1}, together with the discrete logarithm x_i in base g of z_i, unless the corresponding encryption queries have been asked. Our simulation does not make any encryption, x_i is not initialized

if z_i^* has been obtained has a ciphertext by the adversary: This happens with negligible probability. Once we have x_i, one can thus conclude by computing the Z_i and Z_{i+1}, and then h_i, h_{i+1} and X_i. Such a change in the process does not alter the view of the adversary, in any way, since this is a purely syntactic rewriting, with negligible probability of failure.

Corruptions. If P_i gets corrupted, the simulator has to provide its internal state (in particular its private exponent x_i) to $\mathcal{A}$. This is actually the discrete logarithm obtained during the simulation of the decryption oracle on z_i^*.

$\boldsymbol{G}_4$ is thus statistically indistinguishable from $\boldsymbol{G}_3$. Note that we do not use the password before round 2. We will now try to avoid to use it before a corruption.

Game $\boldsymbol{G}_5$: In this game, we first deal with the passive case, in which all the flows are oracle-generated. The simulation of the first three rounds is done as in $\boldsymbol{G}_4$ and we then consider two cases. If all the flows were oracle-generated up to round 4, we require $\mathcal{S}$ to simulate the end of the execution of the protocol on behalf of all the players. Otherwise, it simply follows the protocol (as above).

In the former case (with only oracle-generated flows up to round 4), the simulator starts round 4 by asking a `SamePwd`-query. Notice that since we are assuming that $\mathcal{S}$ knows all the passwords, this boils down to verify that all the passwords are actually the same. Now we distinguish two cases: If all passwords are the same, $\mathcal{S}$ asks a `Delivery` query with a random key and keyword `yes` to the functionality for each player. Otherwise, all the players receive an error message. A problem only appears if $\mathcal{A}$ asks the corresponding query to $\mathcal{H}_0$, but then it must have obtained the values h_i and event `AskH`, as described below, appears.

Remark 1. Note that if all the players have the same password, such a strategy makes this game (almost) indistinguishable with respect to previous one. If, conversely, the players do not have all the same password, they will end-up computing different $Auth_i$'s, thus getting an error message with overwhelming probability. However, a problem may arise if the adversary asks the correct queries to $\mathcal{H}_2$, which is the only way, given all the X_i, to have enough information about all the h_i. Indeed, all the X_i form $(n-1)$ independent linear combinations of the n variables h_i. Fixing one h_{i_0} to a random value is another independent equation. More precisely, we have a unique solution $(h_1, \ldots, h_n)$ to this system of n equations and n unknown, whatever value we give to this h_{i_0}. This means that the value of h_{i_0} is unpredictable and cannot be guessed without having asked the correct query to $\mathcal{H}_2$. However, such an event happens only with negligible probability, we call it `AskH`:

> `AskH`: $\mathcal{A}$ queries the oracle $\mathcal{H}_2$ on the value r where there exists i such that $r = CDH_g(\mathcal{D}_{pw_i}^{ssid,i-1}(z_{i-1}^*), \mathcal{D}_{pw_i}^{ssid,i}(z_i^*))$ and there is no record of the form $(pw_i, (ssid, i-1), *, *, \mathcal{E}, z_{i-1}^*)$ or $(pw_i, (ssid, i), *, *, \mathcal{E}, z_i^*)$ in the list Λ_ε.

The condition on the encryption queries is important: if z_{i-1}^* was obtained from an encryption query, the adversary would know the secret exponent and could then compute the CDH in a trivial way. Since this probability of `AskH` is negligible under the CDH assumption (see proof in Appendix C.2), $\boldsymbol{G}_5$ and $\boldsymbol{G}_4$ are (computationally) indistinguishable.

Game $\boldsymbol{G}_6$: In this game, we suppose that $\mathcal{S}$ still knows the players' passwords and we consider two cases. Either some party has already been corrupted before round 2, then the simulator executes the whole protocol honestly, as in $\boldsymbol{G}_4$. Or no corruption occurred before round 2, we will modify the simulation in order not to use the passwords anymore. Their knowledge is still needed in the first case (which will be dealt with in $\boldsymbol{G}_7$). We focus on the second case in this game: No corruption before the second round.

$\mathcal{S}$ starts the protocol by sending, on behalf of each non-corrupted player P_i, random values c_i, z_i^* and VK_i, as in the previous game. If no corruption occurred, it then starts round 2 by asking a `SamePwd` query. For simplicity we consider two cases here depending on whether it gets answer 'yes' or 'no'. Recall that the players can only get a shared valid key in the first case; otherwise, they will get an error.

Case 1 – They all Share the Same Password ($\boldsymbol{G}_{6a}$): The simulator sends a random value c'_i on behalf of each player P_i. Once all values have been sent, it chooses at random the values h_i^L and h_i^R, with the constraints that $h_i^L = h_{i-1}^R$ and $h_i^R = h_{i+1}^L$ (this is to ensure that $X_1 \oplus \ldots \oplus X_n$ will be equal to 0 as it has to). Finally, it programs the oracle $\mathcal{H}_4$ so that $\mathcal{H}_4(ssid', X_i, i) = c'_i$. This simulation only fails if $\mathcal{A}$ has already asked the query to the oracle for at least one X_i, which probability is bounded by $q_{h_4}/2^{\ell_2}$. At this stage every list contains at least the following values: $(\perp, SK_i, \perp, \perp, z_i^*, c_i, \perp, \perp, h_i^R, h_{i+1}^L, X_i, c'_i)$.

Corruptions. If a player P_i gets corrupted afterwards, the simulator learns its password pw_i and asks for the decryption queries on z_i^*, z_{i-1}^*, and z_{i+1}^* with key pw_i and the appropriate tweaks, as in $\boldsymbol{G}_4$, recovering x_i with overwhelming probability. $\mathcal{S}$ can then compute all the values in the list and provide $\mathcal{A}$ with the internal state of the player. $\mathcal{S}$ finally programs the random oracle $\mathcal{H}_2$ so that the computed values Z_i^R and Z_{i+1}^L are consistent with the h_i^R and h_{i+1}^L already sent, i.e. $\mathcal{H}_2(Z_i^R) = h_i^R$ and $\mathcal{H}_2(Z_{i+1}^L) = h_{i+1}^L$. Event `AskH` can make this programming fail, see Remark 1.

Since we are assuming all players to have the same password (only one connected component), by a similar reasoning we can fill the lists of each remaining player in exactly the same way (and with the same, negligible, probability of failure).

Note that if a player receives a non-oracle-generated X_i, then it will receive a non-oracle-generated $\mathcal{A}uth_i$ and σ_i, which will be refused, due to $\boldsymbol{G}_3$. Similarly, if a player receives a non-oracle-generated $\mathcal{A}uth_i$, it will refuse the corresponding signature.

In round 3, the simulator asks honestly the queries to $\mathcal{H}_1$, in order to provide the players with correct authenticators, and computes (honestly) the corresponding signatures. In round 4, the simulator asks a `Delivery` query with a random key to the functionality for each player. The keyword `b` is set to `yes` except for the players who received non oracle-generated values of the authenticators. If a player was corrupted before, $\mathcal{S}$ gets back the session key and can thus reprogram the oracle $\mathcal{H}_0$ accordingly. Otherwise, event `AskH` must have appeared if $\mathcal{A}$ asks the corresponding query to $\mathcal{H}_0$, as in the former game.

Corruptions. Note that in case of corruptions in rounds 3 or 4, the simulator can easily compute the internal state as above, by only programming $\mathcal{H}_2$.

Case 2 – Some Players Have Different Passwords ($\boldsymbol{G}_{6b}$): The simulator provides each player P_i with an independently-chosen random c'_i. When all values have been sent, it gives them a value X_i chosen at random and programs $\mathcal{H}_4$ in order that $c'_i = \mathcal{H}_4(ssid', X_i, i)$. This programming fails with probability $q_{h_4}/2^{\ell_2}$. (Note that as in the real game, it can happen that $\sum X_i = 0$ with negligible probability.) We then fill in the lists with z_i^* and X_i: all other values are unknown to the simulator. The corruptions are dealt as in Case 1 (this time however there will be some index j for which $h_j^R \neq h_{j+1}^L$ or $h_j^L \neq h_{j-1}^R$).

In round 3, the simulator provides the players with independently-chosen random values for the authentication queries and the corresponding correct signatures. Finally, all the players are given an `error` message in round 4. The corruptions are dealt with as in the former game.

Game $\boldsymbol{G}_7$: We will now conclude, by dealing with sessions where some corruption happened before the second round. We will try not to make use of the knowledge of the password of the honest players. Due to the use of the split functionality model of [6], we can assume that all the players received the same values during the first round since we split the groups according to the first-round messages. In particular, no flow was modified, and $\mathcal{S}$ can extract the password used in non-oracle-generated `flow(1a)` and `flow(1b)` (corrupted players, from the beginning), granted z_i^* and the list Λ_ε. This extraction is unique granted the collision-restriction included in the simulation (see $\boldsymbol{G}_1$). One can argue that this strategy may fail if $\mathcal{A}$ never asked the corresponding encryption query. However, in such a case, $\mathcal{A}$ has no control over the plaintext (and in particular no idea about the discrete logarithm), and we deal with this case as if $\mathcal{A}$ had an incorrect password with respect to its neighbors (see below).

We compare each corrupted player with the neighbor-sets of "consecutive" honest players belonging to the same connected component (sharing the same password). Considering all the possible situations in which a non-corrupted P_i can be, with respect to its neighbors (see Figure 6), we thus distinguish 6 mutually exclusive situations. In the first scheme of this figure, "$=$" means that the simulator knows that P_i has the same password as the two nearest (on the left and on the right) corrupted players (according to the passwords extracted from the encryption queries), whereas "$\neq$" stands for different passwords and "**?**" for unknown relation between the passwords. Corrupted players are denoted by $\mathcal{A}$. Some cases share the same index to denote the fact that they are actually symmetric and can be studied as a single case.

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots P_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (1) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots P_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (2)

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots P_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (2) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots P_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (4)

$\mathcal{A} \ldots \leftarrow = \rightarrow \ldots P_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (3) $\quad \mathcal{A} \ldots \leftarrow \neq \rightarrow \ldots P_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (5)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots P_i \ldots \leftarrow = \rightarrow \ldots \mathcal{A}$ (3)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots P_i \ldots \leftarrow \neq \rightarrow \ldots \mathcal{A}$ (5)

$\mathcal{A} \ldots \leftarrow ? \rightarrow \ldots P_i \ldots \leftarrow ? \rightarrow \ldots \mathcal{A}$ (6)

---

(1) $Z_0^R \quad Z_1^L{=}Z_1^R \quad Z_2^L{=}Z_2^R \quad Z_i^L{=}Z_i^R \quad Z_{i+1}^L{=}Z_{i+1}^R \quad Z_k^L{=}Z_k^R \quad Z_{k+1}^L{=}Z_{k+1}^R \quad Z_{k+2}^L$
$\rightarrow \underset{X_0}{\mathcal{A}} \longleftrightarrow \underset{X_1}{P_1} \longleftrightarrow \ldots \longleftrightarrow \underset{X_i}{P_i} \longleftrightarrow \ldots \longleftrightarrow \underset{X_k}{P_k} \longleftrightarrow \underset{X_{k+1}}{\mathcal{A}} \leftarrow$

---

(2) $Z_0^R \quad Z_1^L{=}Z_1^R \quad Z_2^L{=}Z_2^R \quad Z_k^L{=}Z_k^R \quad Z_{k+1}^L{\neq}Z_{k+1}^R \quad Z_{k+2}^L$
$\rightarrow \underset{X_0}{\mathcal{A}} \longleftrightarrow \underset{X_1}{P_1} \longleftrightarrow \ldots \longleftrightarrow \underset{X_k}{P_k} \leftarrow \neq \rightarrow P_{k+1} \leftarrow$

---

(3) $Z_0^R \quad Z_1^L{=}Z_1^R \quad Z_2^L{=}Z_2^R \quad Z_k^L{=}Z_k^R \quad Z_{k+1}^L \; ? \; Z_{k+1}^R \quad Z_{k+2}^L$
$\rightarrow \underset{X_0}{\mathcal{A}} \longleftrightarrow \underset{X_1}{P_1} \longleftrightarrow \ldots \longleftrightarrow \underset{X_k}{P_k} \leftarrow ? \rightarrow P_{k+1} \leftarrow$

---

(4) $(4a) \rightarrow \mathcal{A} \longleftrightarrow \ldots \leftarrow \neq \rightarrow \ldots \leftarrow ? \rightarrow P_i \leftarrow \neq \rightarrow \mathcal{A} \leftarrow$ or $(4b) \rightarrow \mathcal{A} \leftarrow \neq \rightarrow P_i \leftarrow \neq \rightarrow \mathcal{A} \leftarrow$

or $(4c) \rightarrow \mathcal{A} \longleftrightarrow \ldots \leftarrow \neq \rightarrow \ldots \leftarrow ? \rightarrow P_i \leftarrow ? \rightarrow \ldots \leftarrow \neq \rightarrow \ldots \longleftrightarrow \mathcal{A} \leftarrow$

---

(5) $Z_0^R \quad Z_1^L{=}Z_1^R \quad Z_2^L{=}Z_2^R \quad Z_{i-1}^L{=}Z_{i-1}^R \quad Z_i^L{\neq}Z_i^R \quad ?$
$\rightarrow \underset{X_0}{\mathcal{A}} \longleftrightarrow \underset{X_1}{P_1} \longleftrightarrow \ldots \longleftrightarrow \underset{X_{i-1}}{P_{i-1}} \leftarrow \neq \rightarrow P_i \leftarrow$

---

(6) $\rightarrow \mathcal{A} \ldots \leftarrow ? \rightarrow P_i \leftarrow ? \rightarrow \ldots \mathcal{A} \leftarrow$

Fig. 6. The different cases of connected components

It is easy to verify that such cases cover the entire spectrum of possibilities. Notice that the simulator can easily determine to which case each player belongs by means of the `GoodPwd` queries already asked. This is because, as soon as $\mathcal{S}$ recovers the password used by some corrupted player, it asks a `GoodPwd` query to the player's two neighbors. For each query, if it returns "no", $\mathcal{S}$ stops. Otherwise, it asks a `GoodPwd` query to the next neighbor. This process is iterated until either a query returns "no", or $\mathcal{S}$ ends up having to query a player for which a `GoodPwd` query was already asked (recall that it is not allowed to ask more than one `GoodPwd` query per player). This is why we can have a "**?**", even next to a corrupted player. Note that in the special case where we cannot recover a password used by a corrupted player, we consider (without asking a `GoodPwd` query) that it is in Case (4b) on both sides: the corrupted player does not share the same password with any of its neighbors.

Now we proceed by describing the simulation of round 2 in all the cases. First note that only the first case can lead to a real shared key. In all other cases, at least one password is different from the others, thus the players all receive an error message. But we still have to be able to open correctly the internal states, in case of corruption.

Case 1 ($\boldsymbol{G}_{7a}$): There exists a sequence $P_1, \ldots, P_k$ of players (including P_i we are simulating), between two corrupted players, all sharing the same password (case 1 in Figure 6). If there are more than $n/2$ corrupted players, recall that $\mathcal{A}$ is granted the right to choose the session key. Since $\mathcal{S}$ knows the password of this connected component, it can compute everything correctly. In addition, *before* sending any X_i, it gets back the h_i of the corrupted players due to the c'_i and the random oracle lists, and queries the oracle $\mathcal{H}_0$ to obtain the session key. It then queries the functionality with a `SamePwd`, which either sets the key to this value or to `error`. This way, $\mathcal{S}$ has set the key to the value that $\mathcal{A}$ has chosen by the means of the values sent by the corrupted players. It will be able to reply back to $\mathcal{A}$ with this particular key when the former asks the corresponding query to $\mathcal{H}_0$ – which query is unpredictable to $\mathcal{A}$ until it receives the X_i.

If there are at most $\lfloor n/2 - 1/2 \rfloor$ corrupted players, there exists at least one component of honest players $P_1, \ldots, P_k$ in which $k \geq 2$. This is the key point of the generalization presented in Appendix B.2. In the components where $k = 1$, since the simulator knows the password of P_1, it can simulate it perfectly. In the components where $k \geq 2$, the simulator thus provides the players with independently chosen random c'_j. Once step $(2a)$ is finished, it computes honestly the values h_1^L and h_k^R but chooses at random all the other ones, with the constraint that $h_i^L = h_i^R$ everywhere iteratively (see Remark 1). As before, $\mathcal{S}$ also programs the oracle $\mathcal{H}_4$ so that $\mathcal{H}_4(ssid', X_j, j) = c'_j$, with negligible probability of failure. Before sending any X_i, $\mathcal{S}$ asks a `SamePwd` and a `Delivery` query for a corrupted player, with a random key and keyword `yes`, this way getting either the value of the key or an `error`. If a key was set, $\mathcal{S}$ recovers the h_i of the corrupted players and programs $\mathcal{H}_0$ to this value. This programming only fails if `AskH` appears. In this component, all the X_j are thus unpredictable to the adversary. At this stage every list contains at least the following values: $(\perp, \perp, \perp, z_i^*, c_i, \perp, \perp, h_i^R, h_{i+1}^L, X_i, c'_i)$. In case of later corruption, one proceeds as in $\boldsymbol{G}_{6a}$.

Cases 2 and 3 ($\boldsymbol{G}_{7b}$): There exists a sequence $P_1, \ldots, P_k$ of players (including P_i we are simulating), next to a corrupted player, all sharing the same password, but not necessarily with P_{k+1} (the sequence of successful `GoodPwd` queries stopped with it or a `GoodPwd` query already failed from the right side of P_{k+1}). See cases 2 and 3 in Figure 6.

Again, the simulation is perfect for the k first players, since the simulator knows the password shared among $P_1 \ldots P_k$. Then, $\mathcal{S}$ sends a random X_{k+1} on behalf of P_{k+1}.

In case of later corruption, we point out that in round 1, the two (not yet corrupted) players P_k and P_{k+1}, sent the values z_k^* and z_{k+1}^* without having asked any encryption query. This means that there is a negligible probability that $\mathcal{A}$ has asked the corresponding encryption query with tweaks $(ssid, k)$ and $(ssid, k+1)$, respectively. In fact, querying $\mathcal{H}_2(Z_{k+1})$ in this context is exactly event `AskH`, which means that $\mathcal{A}$ cannot distinguish X_{k+1} from the exact value, and the later programming will succeed, with all but with negligible probability, as in $\boldsymbol{G}_5$.

Cases 4, 5 and 6 ($\boldsymbol{G}_{7c}$): P_i doesn't share the same password with any of its two nearest corrupted neighbors (see Case 4 in Figure 6), or with one of them (see Case 5), or we do not know (Case 6). In this case, we let X_i be random, but nothing else is known except z_i^* and X_i for P_i. First note that P_i is uncorrupted and has sent z_i^* without asking any encryption query. First case, if P_i's neighbor (say P_{i+1}) is a corrupted player, the latter did not ask an encryption query resulting in z_{i+1}^* with password pw_i and tweak $(ssid, i)$ (simply because it queried the encryption oracle with pw, tweak $(ssid, i+1)$, and we are assuming $pw_i \neq pw$). Second case, if P_{i+1} is an honest player, then it must have sent z_{i+1}^* without having invoked the encryption oracle. Thus, in both cases, $\mathcal{A}$ can realize that X_i was not properly chosen only if `AskH` occurs. Later corruptions are dealt with as in $\boldsymbol{G}_5$.

Last Steps: These are dealt with exactly as in the previous game, according to the `SamePwd`-query answer. Only the first case must be dealt with care: $\mathcal{S}$ does not give a random key, but the key obtained by the first `Delivery` query asked above.

Thus, ignoring the event `AskH`, whose (negligible) probability is computed in Appendix C.2, $\boldsymbol{G}_7$ and $\boldsymbol{G}_6$ are indistinguishable.

Game $\boldsymbol{G}_8$: This game is almost the same as the previous one, except that we formalize the behavior of the simulator by introducing the queries to the functionality, in place of the `GoodPwd`, `SamePwd` and `Delivery`-queries. More precisely, we only replace the hybrid queries `GoodPwd` and `SamePwd` with their ideal equivalents. Informally, $\mathcal{S}$ behaves not according to the messages sent, but to the messages received (probably modified by the adversary). In round 1, $\mathcal{S}$ sends a random z_i^* on behalf of each non-corrupted player. For round 2, see games $\boldsymbol{G}_6$ and $\boldsymbol{G}_7$. In round 3, $\mathcal{S}$ asks key delivery queries with a random key, or the key obtained before, see $\boldsymbol{G}_{7a}$. In round 4, $\mathcal{S}$ sets `b` to `no` for the players receiving a non-oracle generated flow in round 3, and to `yes` for the others. If a session aborts or terminates, then $\mathcal{S}$ reports it to $\mathcal{A}$. We now show that $\boldsymbol{G}_8$ is indistinguishable from the ideal game. Say that the players have matching sessions if they share the same $ssid'$ (which implies that they share the same VK_i and z_i^* due to the use of the split functionality).

It is clear that players sharing the same password will obtain a random key, both in $\boldsymbol{G}_8$ (from $\boldsymbol{G}_5$) and IWE (except for players receiving non-oracle generated flows, modeled by the bit `b`). This key will be not chosen by the adversary unless there are enough corrupted players (since there will always be two honest players next to each others, see $\boldsymbol{G}_7$). Finally, players not sharing the same password will receive an error. Now, we need to show that two players will receive the same key in $\boldsymbol{G}_8$ if and only if it happens in IWE.

This is clearly the case for players with matching session (with or without the same password). This follows from $\boldsymbol{G}_5$ or $\boldsymbol{G}_7$ in the real world, and from the `NewSession` queries mentioning the same group of players in the ideal world. Finally, consider the case of players with no matching sessions. It is clear that in $\boldsymbol{G}_8$ the session keys of those players will be independent because they are not set in any of the games. In IWE, the only way that they receive matching keys is that the functionality receives two `NewSession` queries with the same $ssid'$ and a group where all players share the same passwords.

C.2 Probability of AskH

We now show that event `AskH` happens with negligible probability, with the help of a reduction to the CDH problem, given one CDH instance (U, V). At the beginning of the game, globally for all sessions, $\mathcal{S}$ chooses two decryption queries. With probability $1/(q_{\mathcal{D}}^2)$, $q_{\mathcal{D}}$ being the number of decryption queries, these queries will correspond to those really asked by the adversary. For these two queries, we use U and V to simulate them. More precisely, for the first decryption query z_i^*, if $(*, (ssid, i), *, *, *, z_i^*) \in \Lambda_\varepsilon$, abort. Otherwise, add $(pw, (ssid, i), U, \perp, \mathcal{D}, z_i^*)$ to the list. For the second one, z_j^*, if $(*, (ssid, j), *, *, *, z_j^*) \in \Lambda_\varepsilon$, abort. Otherwise, add $(pw, (ssid, j), V, \perp, \mathcal{D}, z_j^*)$ to the list.

Let `AskH`$'$ be the event in which $\mathcal{A}$ has queried the random oracle $\mathcal{H}_2$ on the value

$$CDH(\mathcal{D}_{pw}^{(ssid,i)}(z_i^*), \mathcal{D}_{pw}^{(ssid,j)}(z_j^*)).$$

This implies that $|\Pr(\texttt{AskH}')| = q_{\mathcal{D}}^2\, q_{\mathcal{H}_2}\, |\Pr(\texttt{AskH})|$, $q_{\mathcal{H}_2}$ being the number of queries to the random oracle $\mathcal{H}_2$. As we know that $CDH(z_i, z_j) = CDH(U, V)$, we finally note that if $\mathcal{A}$ raises event `AskH`, it has solved the CDH problem, which happens only with negligible probability.